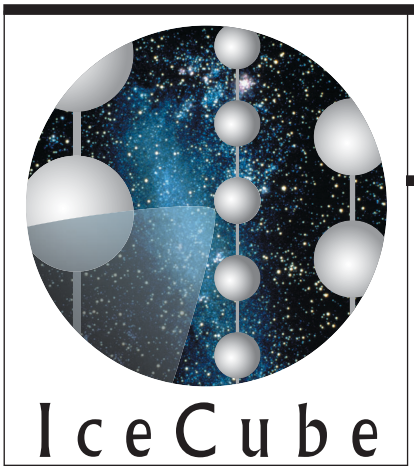


IceCube Software

File Organization for C and C++ files in the IceCube Software Development Environment

Simon Patton L.B.N.L.

This documents outline how C and C++ files should be organized so that they can be used within the IceCube Software Development Environment.

Last Modified: February 18, 2004 3:37 pm

Version:0.0

1.0 Introduction

1.1 Purpose

The purpose of this document is to describe how C and C++ files should be organized so that they can be used within the IceCube Software Development Environment.

The intended audience of this document is C and C++ code developers.

1.2 Scope

The scope of this document is to explain where and why various different C and C++ files should be placed within the structures used within the IceCube Software Development Environment.

1.3 References

The following references are used throughout this document.

[IceSoft-xxx] An Introduction to the IceCube Software Development Environment.

1.4 Definitions and Acronyms

package: A organizational groups of conceptually related files.

platform: A combination of the operating system and computer architecture for which a particular build of the code is produced.

project: A collection of files and resources that are all packaged into a single library file.

1.5 Overview of this Document

The next section...

2.0 Organizational Sub-division of Code

The basic organizational sub-division of code in the IceCube software system is the **project**. All C and C++ files within are project are targeted to the creation of a single library file or, in the case where Java code is also included in the project, a single library and single jar file. The name of a project should be unique with IceCube and clearly identify the responsibilities that the project has assigned.

Within a single project the code can be further divided into packages. If there is more than one package within a project then the contents of a package is more tightly related than the overall contents of the enclosing project. The idea of packages has been taken for the work done on Java development within the Software Environment but, unlike Java where a package provides a naming and accessibility scope, in C and C++ the idea of a package is simply an organizations division of the code that does not provide any special functionality.

External access to the header file contained in a project is based upon the package which contained the header file (See 2.0 ‘‘Organizational Sub-division of Code’’). In most cases a the main interface of a project will be a package with the same name as the project. However despite this, it is important to remember that the package is the unit of access. One product of this approach is that the package name within a project should be unique across *all* package names with the IceCube software.

Within a project, files are divided into two branches; “public” and “private”. The public branch of a project contains those files which are available for other projects to use. These are mostly expected to be include files which define the public interface of the project. The private branch of a project contains all the other files needed to realize the project.

3.0 Include Files

The specification of an include file from with a piece of code needs to be unique within all of IceCube’s software. However the task of ensuring that all include files names are unique is a daunting, and in the long run impractical one. However by requiring the name of a package (a much larger unit of granularity) to be unique and then having include files specified by `<package>/<file>` then this specification will be unique within the IceCube software. Moreover experience shows that, especially during early development, packages often get moved between projects and adoption of this style of specification means that no code need be modified as a result of this type of transfer. This means that all `include` statements should appear in the following form:

#include "<package>/<file>"

The placement of an include file, within the structure of a project, is dictated by a number of considerations.

- The package to which the file is assigned.
- Is the file part of the public interface of the project?
- Are the contents of the file written of a specific **platform**.[1].
- Is the file generated as part of the build process.

Each of these considerations defines part of the path in which the file will be placed.

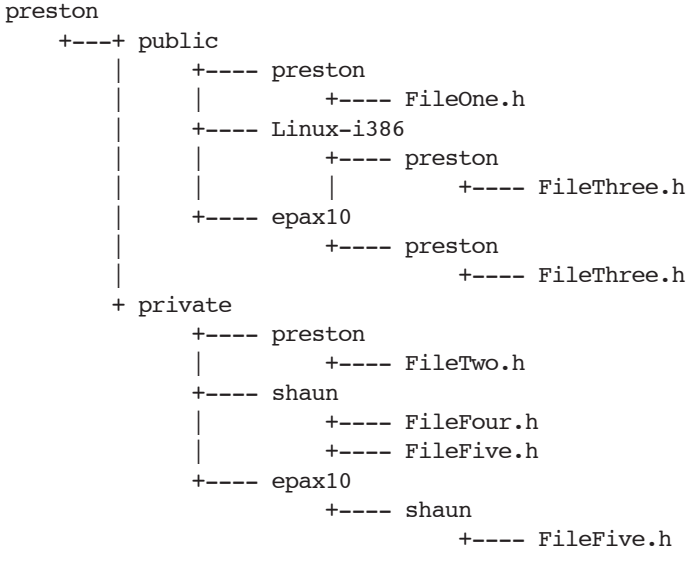
The easiest way to demonstrate how these considerations define a file’s path is to look at some examples and then explain the details of how these paths are constructed. Figure 1 shows the storage location for the following example files.

- `FileOne.h`, `FileTwo.h` and `FileThree.h` belong to `preston` package in the `preston` project.
- `FileFour.h` and `FileFive.h` belong to `shaun` package in the `preston` project.

1. A platform is defined as the combination of the operating system and computer architecture for which a build is produced.

- `FileOne.h` and `FileThree.h` are part of the public interface of the `preston` project.
- `FileTwo.h`, `FileFour.h` and `FileFive.h` are part of the implementation of the `preston` project and thus in its private portion.
- The contents of `FileThree.h` depends on the target platform of the build. There are version of this file for Linux running on i386 boxes, i.e. to DOM simulation, and also for the Excalibur evaluation board. This file does not exist for any other platfrom.
- `FileFive.h` is also platform dependent, but in this case only the version used on the Excalibur evaluation board needs to be radically different, all other target platforms can use a single file.

FIGURE 1: The layout of the include files specified in the text.



(The auto-generated include files are still under development and so are not shown in these examples.)

As can be seen in Figure 1, in a normal project the first division of files is into `public` and `private` directories. This allows the public interface to be published by simply making the `public` directory available to other projects. Within either of these directories files whose contents can be built on any platform are stored in a sub-directory whose name it that of the package to which the file is assigned. This supports the `<package>/<file>` specification used by `#include` statements. However if the contents of a file can only be built on a specific platform then it needs to be stored in a package sub-directory which itself lives in sub-directory of the relevant `public` or `private` directories that specifies the platform on which that file should be used.

Given this directory structure for files, the search order for include files is the following:

1. generated files created by this project.

2. public, platform specific files (to make sure external projects and internal files agree.) within this project.
3. private, platform specific, files within this project.
4. public, generic (i.e. non-platform specified), files within this project.
5. private, generic, files within this project.
6. generated files created by projects upon which this project is dependent.
7. public, platform specific, files in projects upon which this project is dependent.
8. public, generic (i.e. non-platform specified), files in projects upon which this project is dependent.

4.0 Source Files

Source code files should only ever contain implementation details for a project. Therefore the source code files should appear in the `private` branch of the project. As these files, like include files, are assigned to packages with a project they too should be stored in a sub-directory which is named after their package. These package sub-directories can either appear as sub-directories of the `private` directory if their contents can be used to build code on any platform, or as sub-directories of a platform specific subdirectory.

Figure 2 shows the layout the source files that correspond to the include files specified earlier.

FIGURE 2: The layout of the source files that correspond to the include files specified in the text.

```
preston
    +---+ private
              +---- preston
              |         +---- FileOne.c
              |         +---- Linux-i386
              |         |         +---- FileThree.c
              |         +---- epax10
              |                   +---- FileThree.c
              +---- shaun
                        +---- FileFour.c
                        +---- FileFive.c
                        +---- epax10
                                  +---- FileFive.c
```

The standard makefile for "C" code will search for source files in the following order:

1. private, platform specific, files within this project.
2. private, generic, files within this project.